Many slides from this lecture are adapted from:
• Slides with the book “Heterogeneous Computing with OpenCL 2.0” 3rd edition
A Quick Refresh
OpenCL Platform Model

Processing Element

Compute Unit

Compute Device

Host
A Bit of Vocabulary

- **Kernel**: Smallest unit of execution, like a C function
- **Host program**: A collection of kernels
- **Work group**: A collection of work items
  - Has a unique work-group ID
  - Work items can synchronize
- **Work item**: An instance of kernel at run time
  - Has a unique ID within the work-group
OpenCL Memory Model

- Relaxed consistency model
- Implementations map this hierarchy to available memories
Big Picture

OpenCL Context

OpenCL Program objects

Program object

Kernel # 0  Kernel # 1  ....

Memory objects

Events

Buffers

Images

Pipes

Host-side command queues

Device Queue

CPU Device

GPU Device

Device Queue
Case Study: Image Clustering
Problem definition

• The bag-of-words (BoW) model:
  – one of the most popular approaches to image classification.
  – Treats an image’s features as words.
  – Represents the image as a vector of the occurrence counts of image features (words).

• We will discuss:
  – OpenCL implementation of the histogram builder → a very important component of BoW.
SURF = Speed-UP Robust Feature

- Features are quantized
- Typically by k-means clustering
- And mapped into clusters.
- The centroid of each cluster is known as a visual word
Histogram Builder

• **GOAL:** Determine to which centroid each descriptor belongs.

• **Input data:** Both the descriptors of the SURF features and the centroid have 64 dimensions.

• **Method:**
  1. Compute the Euclidean distance between the descriptor and all the centroids.
  2. Assign each SURF descriptor to its closest centroid.

• **Output:** The histogram is formed by counting the number of SURF descriptors assigned to each centroid.
// Loop over all the descriptors generated for the image
for(int i = 0; i < n_desc; i++)
{
    membership = 0;
    min_dist = FLT_MAX;
    // Loop over all the cluster centroids available
    for(j = 0 ; j < n_cluster; j++)
    {
        dist = 0;
        // n_features: No. of elements in each descriptor (64)
        // Calculate the distance between the descriptor and the centroid
        for(k = 0 ; k < n_features; k++)
        {
            dist_temp = surf[i][k] - cluster[j][k];
            dist += dist_temp * dist_temp;
        }
        // Update the minimum distance
        if(dist < min_dist)
        {
            min_dist = dist;
            membership = j;
        }
    }
    // Update the histogram location of the closest centroid
    histogram[membership] += 1;
}
Second: Parallel CPU Implementation with OpenMP

```c
// All the descriptors for the image can be handled in parallel
#pragma omp parallel for schedule(dynamic)
for(int i = 0; i < n_desc; i++)
{
    membership = 0
    min_dist = FLT_MAX
    for(j = 0 ; j < n_cluster; j++)
    {
        dist = 0;
        // n_features: No. of elements in each descriptor (64)
        // Calculate the distance between the descriptor and the centroid
        for(k = 0 ; k < n_features; k++)
        {
            dist_temp = surf[i][k]−cluster[j][k];
            dist += dist_temp * dist_temp;
        }
        // Update the minimum distance
        if(dist < min_dist)
        {
            min_dist = dist;
            membership = j;
        }
    }
    // The histogram needs to be updated atomically since multiple
    // descriptors could update the same element
    #pragma omp atomic
    histogram[membership] += 1
}
```
Second: GPU Implementation #1 with OpenCL

Atomically:
• fetch data pointed to by &histogram[]
• Add 1 to it
• Store the result back to &histogram[]

```c
int desc_id = get_global_id(0);

int membership = 0;
float min_dist = FLT_MAX;

// For each cluster, compute the membership
for (int j = 0; j < n_centroids; j++) {
    float dist = 0;

    // n_features: No. of elements in each descriptor (64)
    // Calculate the distance between the descriptor and the centroid
    for (int k = 0; k < n_features; k++) {
        float temp = descriptors[desc_id*n_features+k] -
                     centroids[j*n_features+k];
        dist += temp*temp;
    }

    // Update the minimum distance
    if (dist < min_dist) {
        min_dist = dist;
        membership = j;
    }
}

// Atomic increment of histogram bin
atomic_fetch_add_explicit(&histogram[membership], 1,
                          memory_order_relaxed, memory_scope_device);
```
Second: GPU Implementation #1 with OpenCL

```c
__kernel
void kernelGPU1(
    __global float *descriptors,
    __global float *centroids,
    __global int *histogram,
    int n_descriptors,
    int n_centroids,
    int n_features)
{
    // Global ID identifies SURF descriptor
    int desc_id = get_global_id(0);
    int membership = 0;
    float min_dist = FLT_MAX;

    // For each cluster, compute the membership
    for(int j = 0; j < n_centroids; j++) {
        float dist = 0;

        // n_features: No. of elements in each descriptor (64)
        // Calculate the distance between the descriptor and the centroid
        for(int k = 0; k < n_features; k++) {
            float temp = descriptors[desc_id*n_features+k] -
                         centroids[j*n_features+k];
            dist += temp*temp;
        }

        // Update the minimum distance
        if(dist < min_dist) {
            min_dist = dist;
            membership = j;
        }
    }

    // Atomic increment of histogram bin
    atomic_fetch_add_explicit(&histogram[membership], 1,
                              memory_order_relaxed, memory_scope_device);
}
```

Remember:
- Successive work items are executed in lock-step

What do you think about the memory access of this line?
Suppose we have 4 work-items in a workgroup. They have IDs: 0, 1, 2, and 3. 
n_features = 64
Let's take the first iteration of k (i.e. k = 0)
Those work items will access:
- descriptors[0]
- descriptors[64]
- descriptors[128]
- descriptors[192]

Pretty Bad.. Indeed!

Cannot be coalesced 😞
We need to change the way descriptors[] is accessed

A small kernel where each thread reads one element of the input matrix from global memory and writes back the same element at its transposed index in the global memory. Called before the main kernel of the histogram.

Transpouse
We need to change the way descriptors[] is accessed

A small kernel where each thread reads one element of the input matrix from global memory and writes back the same element at its transposed index in the global memory. Called before the main kernel of the histogram.

Transpose
Now we make a small change in the main kernel

```c
21 // n_features: No. of elements in each descriptor (64)
22 // Calculate the distance between the descriptor and the centroid
23 for (int k = 0; k < n_features; k++) {
24     float temp = descriptors[k*n_descriptors+desc_id] -
25                     centroids[j*n_features+k];
26     dist += temp*temp;
27 }
```
Another Optimization

for(int k = 0; k < n_features; k++) {
    float temp = descriptors[k*n_descriptors+desc_id] — 
                  centroids[j*n_features+k];
    dist += temp*temp;
}

• Data in these buffers are accessed multiple times
• centroids[] access is independent of thread ID
Local Memory
(From OpenCL Perspective)

• Local memory is a high bandwidth low-latency memory used for sharing data among work-items within a work-group.

• However:
  - local memory has limited size
  - on AMDRadeon HD 7970 GPU there is 64 KB of local memory per compute unit, with the maximum allocation for a single work-group limited to 32 KB.
Local memory is primarily intended to allow communication of data between work items, and no data is shared when accessing descriptors!

**BUT:**
- For some GPUs (e.g. Radeon HD 7970) local memory is mapped to the **local data shares (LDS)**, which provides four times more storage than the general-purpose level 1 (L1) cache. Placing this buffer in the LDS may provide low-latency access to data that could otherwise result in cache misses.
- Even assuming a cache hit, LDS memory has a lower latency than the L1 cache

**Tradeoff:** the use of local memory will limit the number of in-flight work-groups, potentially underutilizing the GPU’s execution units and memory system.

This is an optimization that needs to be considered on a per-architecture basis.
Local Memory

- On GPUs, local memory maps to a high-bandwidth, low-latency memory located on chip
  - Useful for sharing data among work-items within a work-group
  - Accesses to local memory are usually much faster than accesses to global memory (even cached global memory)
  - Accesses to local memory usually do not require coalescing
  - More forgiving than global memory when having non-ideal access patterns
- Additional advantages on some AMD GPUs (e.g., Radeon HD 7970)
  - Local memory is mapped to LDS, 4x larger than L1 cache
  - LDS has a lower latency than L1 cache
- The tradeoff is that the use of local memory will limit the number of in-flight work-groups
Returns the number of local work-items specified in dimension

// Global ID identifies SURF descriptor
int desc_id = get_global_id(0);
int local_id = get_local_id(0);
int local_size = get_local_size(0);

// Store the descriptors in local memory
  __local float desc_local[4096]; // 64 descriptors * 64 work-items
for(int i = 0; i < n_features; i++) {
  desc_local[i*local_size + local_id] = descriptors[i*n_descriptors + surf_id];
}
barrier(CLK_LOCAL_MEM_FENCE);

int membership = 0;
float min_dist = FLT_MAX;

// For each cluster, compute the membership
for(int j = 0; j < n_centroids; j++) {
  float dist = 0;
  // n_features: No. of elements in each descriptor (64)
  // Calculate the distance between the descriptor and the centroid
  for(int k = 0; k < n_features; k++) {
    float temp = desc_local[k*local_size+local_id] -
                centroids[j*n_features+k];
    dist += temp*temp;
  }
  // Update the minimum distance
  if(dist < min_dist) {
    min_dist = dist;
    membership = j;
  }
}

// Atomic increment of histogram bin
atomic_fetch_add_explicit(&histogram[membership], 1,
memory_order_relaxed, memory_scope_device);
// Store the descriptors in local memory
__local float desc_local[4096]; // 64 descriptors * 64 work-items
for(int i = 0; i < n_features; i++) {
    desc_local[i*local_size + local_id] = descriptors[i*n_descriptors + surf_id];
}
barrier(CLK_LOCAL_MEM_FENCE);
Mapping the centroids[] buffer to constant memory is as simple as changing the parameter declaration from __global to __constant
**Constant Memory**

- **Constant memory** is a memory space to hold data that is accessed simultaneously by all work-items
  - Usually maps to specialized caching hardware that has a fixed size
- **Advantages for AMD hardware**
  - If all work-items access the same address, then only one access request will be generated per wavefront
  - Constant memory can reduce pressure from L1 cache
  - Constant memory has lower latency than L1 cache
Performance on AMD Radeon 7970

<table>
<thead>
<tr>
<th>No. of Features</th>
<th>Transform Kernel (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.05</td>
</tr>
<tr>
<td>16,384</td>
<td>0.50</td>
</tr>
<tr>
<td>65,536</td>
<td>2.14</td>
</tr>
</tbody>
</table>

The small kernel to make the transpose
# Performance on AMD Radeon 7970

<table>
<thead>
<tr>
<th># of Clusters</th>
<th># of SURF Descriptors</th>
<th>GPU Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GPU1</td>
</tr>
<tr>
<td>8</td>
<td>4096</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>16,384</td>
<td>3.60</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>15.36</td>
</tr>
<tr>
<td>16</td>
<td>4096</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>16,384</td>
<td>7.10</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>30.41</td>
</tr>
<tr>
<td>64</td>
<td>4096</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td>16,384</td>
<td>28.28</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>122.09</td>
</tr>
<tr>
<td>128</td>
<td>4096</td>
<td>4.96</td>
</tr>
<tr>
<td></td>
<td>16,384</td>
<td>55.70</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>243.30</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>10.49</td>
</tr>
<tr>
<td></td>
<td>16,384</td>
<td>109.67</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>488.54</td>
</tr>
</tbody>
</table>

Kernel running time in ms.

- **GPU1**: original kernel
- **GPU2**: using transpose
- **GPU3**: vector (we did not cover)
- **GPU4**: transpose + local mem
- **GPU5**: transpose + local mem + constant mem
Let’s Summarize Optimizations
Coalescing Memory Accesses

- AMD hardware, 64 work-items form a wavefront and must execute the same instruction in a SIMD manner (CUDA Warp counterpart)

- For the AMD R9 290X GPU, memory accesses of 16 consecutive work-items are evaluated together and can be coalesced to fully utilize the bus
  - This unit is called a quarter-wavefront and is the important hardware scheduling unit for memory accesses
**Coalescing Memory Accesses**

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an AMD R9 285 GPU.
Vectorization

- Vectorization allows a single work-item to perform multiple operations at once
- Explicit vectorization is achieved by using vector datatypes (such as float4) in the source program
  - When a number is appended to a datatype, the datatype becomes an array of that length
  - Operations can be performed on vector datatypes just like regular datatypes
    - Each ALU will operate on different element of the float4 data
- IMPORTANT: CPUs and previous generations of AMD GPUs benefit from explicit vectorization
  - Current generations of AMD and NVIDIA GPUs execute “scalar” operations on SIMD lanes, which do not benefit from explicit vectorization
Vectorization

- Vectorization improves memory performance on AMD Northern Islands and Evergreen GPUs

```c
__kernel void
Copy4(__global const float4 * input,
     __global float4 * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return;
}

__kernel void
Copy1(__global const float * input,
     __global float * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return;
}
```
Occupancy

• Work-items from a work-group are launched together on a compute unit

• If there are enough resources available, multiple work groups can be mapped to the same compute unit at the same time
  – Wavefronts from multiple work-group can be swapped in to hide latency

• Resources are fixed per compute unit (number of registers, local memory size, maximum number of wavefronts)
  – Any one of these resource constraints may limit the number of work-groups on a compute unit
Conclusions

• Although writing a simple OpenCL program is relatively easy, optimizing code can be more difficult.

• Occupancy is affected by:
  – registers in computer unit
  – local memory in compute unit
  – # work items per work group

• By using different mappings (using `get_local_id()`, `get_global_id()`, ...), the same thread can be assigned to access different data elements.